Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Dal Paradigma Funzionale a Quello Logico in Presenza di Scelte Probabilistiche: un Approccio Basato sulla Geometria dell'Interazione

Relatore:
Chiar.mo Prof.
Ugo Dal Lago

Presentata da:
Giulio Vaccari

II Sessione
Anno Accademico 2017/2018

Alla mia famiglia

Introduzione

La principale attività svolta durante la scrittura di un programma informatico consiste nel riuscire a catturare la realtà di interesse per un dato problema, in modo tale da poterla esprimere nei termini formali di un linguaggio di programmazione. Ogni linguaggio possiede una propria filosofia sul come definire e risolvere i problemi, e quando programmiamo questa ci influenza spingendoci a guardare la nostra realtà attraverso un particolare punto di vista. A prescindere dal linguaggio in cui è scritto, un programma può essere visto come un oggetto formale che nella sua struttura racchiude intrinsecamente il problema che intende risolvere e la realtà a lui associata. Questa struttura logica può essere reinterpretata sotto nuove angolazioni, permettendoci di vedere lo stesso programma sotto una luce completamente diversa, traendone sorprendenti vantaggi.

In questa tesi verrà trattato lo sviluppo di un software che svolge la funzione di traduttore tra due linguaggi di programmazione. Lo scopo di un traduttore è quello di trasformare un programma scritto in un dato linguaggio in un nuovo programma funzionalmente equivalente a quello di partenza ma scritto in un linguaggio diverso. Il lavoro svolto da un traduttore concettualmente è tanto più interessante tanta è la differenza tra i due linguaggi tra cui effettua la traduzione, e nel nostro caso, i due linguaggi che tratteremo sono cosi diversi da appartenere a due paradigmi di programmazione distinti.

Il primo capitolo della tesi introdurrà il linguaggio da cui parte la traduzione, ossia il lambda calcolo probabilistico. Vedremo per prima cosa il paradigma di programmazione funzionale su cui si basa, per poi studiare la

struttura dei programmi definibili nel linguaggio. La versione del lambda calcolo che utilizzeremo sarà inoltre provvista di un sistema di tipi, che rivestirà un ruolo centrale nel processo di traduzione. Il probabilismo farà la sua comparsa attraverso gli "atomi probabilistici", che permetteranno la manipolazione di dati il cui valore non è determinato ma dipendente da un valore di probabilità.

Nel secondo capitolo studieremo ProbLog, un linguaggio fondato sul paradigma di programmazione logica arricchito con costrutti probabilistici, che rappresenterà il linguaggio di destinazione per il traduttore. I linguaggi logici permettono un approccio alla programmazione basato sulla definizione di teorie logiche, in cui da proposizioni assunte vere si derivano nuovi risultati attraverso un processo di deduzione formale. La caratteristica principale di ProbLog che lo differenzia dagli altri linguaggi logici risiede nella possibilità di definire proposizioni che risultano vere con una data probabilità, permettendoci di modellare realtà in cui sono presenti fatti e regole non più per forza veri in senso assoluto.

I costrutti probabilistici posseduti dai nostri due linguaggi permettono loro un approccio completamente diverso alla risoluzione dei problemi, abbandonando l'idea intuitiva secondo cui una computazione debba restituire sempre un unico risultato: nell'ottica probabilistica una computazione può fornirci più esiti possibili, ognuno con una data probabilità di verificarsi.

Lo scopo del terzo capitolo è di introdurre il lettore alla Geometria dell'Interazione, una semantica per la logica lineare introdotta dal logico Jean-Yves Girard, che viene presentata qui in modo molto semplificato e che costituisce il principale strumento teorico impiegato nel processo di traduzione. Il quarto ed ultimo capitolo infine presenterà il traduttore andando nei dettagli della sua implementazione.

Indice

Introduzione **i**

1 Il Lambda Calcolo **1**

1.1 Il Paradigma Funzionale 1

1.2 Il Lambda Calcolo Puro 2

1.3 Lambda Calcolo e Sistemi dei Tipi 5

1.4 L'Aggiunta del Probabilismo 9

2 ProbLog **11**

2.1 Il Paradigma Logico 11

2.2 Prolog 12

2.2.1 I Fatti 12

2.2.2 Le Regole 14

2.3 ProbLog: Logica Probabilistica 16

2.4 La Semantica di ProbLog 17

3 La Geometria dell'Interazione **19**

3.1 Reinterpretare un Programma 20

3.2 Le Regole di Inferenza Secondo La Nuova Semantica 21

3.3 Un Programma Come un Flusso di Dati 27

4 Implementazione Del Traduttore **31**

4.1 Lambda-Termini in Python 31

4.2 Rappresentazione dei Programmi Logici 34

4.3 Il Processo di Traduzione 35
4.3.1 Definizione degli Operatori in ProbLog 35
4.3.2 Numerazione dei Tipi 36
4.3.3 Creazione delle Definizioni per le Costanti Probabilistiche 39
4.3.4 Creazione dei Collegamenti tra Tipi 39
4.3.5 Traduzione delle Regole di Inferenza 43
4.4 Un Esempio di Traduzione 43

Conclusioni **47**

Bibliografia **49**

Capitolo 1

Il Lambda Calcolo

Il linguaggio di partenza per il traduttore in oggetto, è rappresentato da una variante del lambda calcolo. Vedremo come questo linguaggio sia fondato su un paradigma diverso da quelli usuali basati sul modello della macchina di Turing, che verrà trattato nella prima sezione di questo capitolo. Studieremo dunque il linguaggio prima nella sua versione originale, e successivamente in una sua variante provvista di un semplice sistema di tipi. Per finire arricchiremo ulteriormente il lambda calcolo con l'aggiunta di elementi probabilistici, studiando come questi oltre ad incrementarne le funzionalità, modifichino profondamente il concetto stesso di computazione. Per ulteriori approfondimenti riguardo il linguaggio ed i sistemi di tipi, rimando alla lettura di [1].

1.1 Il Paradigma Funzionale

Nell'ottica del paradigma di programmazione funzionale, tutto può essere visto come una funzione matematica. Le funzioni costituiscono gli elementi base del linguaggio, ed un programma è semplicemente il risultato di una composizione di funzioni definite dal programmatore. Una caratteristica fondamentale di questo paradigma, è costituito dalla possibilità di avere funzioni di ordine superiore, ossia che possono accettare come parametro altre fun-

zioni, oltre che restituirne di nuove come risultato. Nella sua versione più pura, l'unico tipo di dato contemplato nel paradigma è la funzione, ma fortunatamente i moderni linguaggi che implementano questo modello sono stati aggiornati per supportare nativamente i tipi primitivi a cui siamo abituati, come ad esempio i numeri interi. Una delle caratteristiche che più colpisce di questo modo di programmare, è senz'altro la mancanza dei costrutti iterativi più comunemente utilizzati, come il `for` ed il `while`. Lo strumento principale a disposizione del programmatore che intende affacciarsi alla programmazione funzionale è la ricorsione, ovvero la possibilità di definire funzioni che nel loro corpo chiamano loro stesse. Questo meccanismo è in grado di sostituire completamente i costrutti iterativi tipici dei linguaggi procedurali, ed è la colonna portante del paradigma. Uno dei vantaggi principali della programmazione funzionale, è il suo distacco dal concetto di "stato" della computazione, a cui ci aveva abituati la macchina di Turing. I paradigmi procedurali infatti, vedono una computazione come una successione di modifiche allo stato della macchina su cui il programma è eseguito, dove con stato intendiamo un insieme di variabili che definiscono la macchina stessa. Nel paradigma funzionale puro, non esiste un concetto di variabile come quello presente nei linguaggi procedurali. Una volta definito un oggetto all'interno di un programma funzionale, questo è immutabile per tutta la durata dell'esecuzione. La modifica dello stato di un oggetto si traduce quindi nella creazione di un oggetto nuovo a partire da quello precedente, ma lasciando il primo inalterato. Questa caratteristica dei linguaggi funzionali li rende particolarmente adatti all'impiego nella programmazione di sistemi paralleli, siccome ogni processo non interferirà coi dati su cui operano gli altri.

1.2 Il Lambda Calcolo Puro

Il lambda calcolo è un sistema formale i cui termini possono essere interpretati come programmi funzionali. Tale sistema possiede un insieme di regole per la costruzione dei termini o programmi, che costituiscono la

grammatica del linguaggio:

```
T :=
    x |          variabile
    λx.T |       astrazione
    T T          applicazione
```

Come possiamo vedere, la grammatica è costituita da tre produzioni:

Variabile: Un lambda-termine può consistere in una singola variabile.

Astrazione: Rappresenta la definizione di una funzione senza nome. La variabile x è un parametro, che può essere utilizzato all'interno del corpo T della funzione, che è anch'esso un termine.

Applicazione: Un lambda-termine può essere il risultato dell'applicazione di un termine ad un altro termine. Il termine a sinistra rappresenta la funzione, mentre quello a destra il suo argomento.

Quella fin qui esposta rappresenta la versione pura del lambda calcolo ma, come già accennato in precedenza, può essere conveniente arricchire il linguaggio per renderlo di più semplice comprensione. Il tipo di dato principale che verrà utilizzato in questa tesi è quello booleano, e per questo motivo aggiungiamo direttamente nella sintassi del lambda calcolo il supporto a questo tipo, ora primitivo del linguaggio:

```
T := ... | tt | ff | not | and | or
```

I valori tt ed ff rappresentano rispettivamente le costanti True e False, mentre le altre funzioni sono gli usuali operatori logici booleani.

Finora abbiamo discusso delle regole formali utilizzate per la costruzione dei nostri programmi, ma non abbiamo parlato della loro esecuzione. La valutazione dei termini avviene per mezzo di due relazioni volte a definire il concetto di "calcolo" di una funzione: l'esecuzione di un programma è ottenuta attraverso la successiva riscrittura del termine iniziale in nuovi termini, seguento ad ongni passo una delle regole seguenti:

$$\alpha\text{-conversione}: \ (\lambda x.T) \equiv_\alpha \lambda y.T\{y/x\}$$

L'α-conversione permette di rinominare la variabile presa come parametro dalla funzione, aggiornando anche i relativi riferimenti alla variabile all'interno del corpo della funzione. Con $T\{y/x\}$ si indica l'operazione di sostituzione della variabile x con la variabile y nel termine T. Esiste una vasta documentazione sulle problematiche riguardanti le name collisions tra variabili e sui sistemi sviluppati per risolverle, ma in questa tesi la questione non verrà particolarmente approfondita. Si rimanda il lettore interessato alla lettura di [1].

$$\beta\text{-riduzione}: \ (\lambda x.T) \ V \rightarrow_\beta T\{V/x\}$$

dove

$$V := tt \mid ff \mid not \mid and \mid or \mid \lambda x.T$$

La β-riduzione rappresenta senz'altro il più importante passo di riscrittura del linguaggio. Rappresenta l'applicazione di funzione, implementata attraverso l'utilizzo della sostituzione. V definisce l'insieme dei "valori" nel lambda calcolo, individuando quei termini che non possono essere ulteriormente riscritti attraverso un'operazione di β-riduzione, e che sono quindi già stati completamente valutati. Concludiamo la definizione della β-riduzione riportando due regole che definiscono la chiusura contestuale di $\rightarrow_\beta$, specifi-

cando ulteriormente come quest'ultima possa essere utilizzata nella riduzione dei termini del linguaggio:

$$\frac{T_1 \rightarrow_\beta T_1'}{T_1\ T_2 \rightarrow_\beta T_1'\ T_2} \qquad \frac{T_2 \rightarrow_\beta T_2'}{T_1\ T_2 \rightarrow_\beta T_1\ T_2'}$$

Le due regole sopra riportate permettono di effettuare la valutazione di un termine costituito dall'applicazione di una funzione T_1 ad un argomento T_2 attraverso la riduzione rispettivamente del primo o del secondo termine. Siccome abbiamo aggiunto dei nuovi elementi sintattici per rappresentare i booleani nei nostri termini, è necessario introdurre le relative regole di valutazione, ma queste non verranno qui riportate.

Per concludere, vediamo ora un semplice esempio di valutazione di un lambda-termine:

$$(\lambda x.\,\mathsf{not}\ x)\ \mathsf{tt} \rightarrow_\beta \mathsf{not}\ \mathsf{tt} \rightarrow_{not} \mathsf{ff}$$

Il programma in questione applica al valore booleano `tt` una funzione che preso un parametro ne restituisce il suo valore negato. La computazione avviene in due passi: nel primo il valore `tt` viene sostituito ad `x` nel corpo della funzione, attraverso l'utilizzo della regola di β-riduzione. Nel secondo passo `not tt` viene riscritto nel termine `ff`, attraverso una delle regole che definiscono la valutazione dell'operatore `not`. L'arresto della computazione è dovuto all'impossibilità di applicare nuove regole di valutazione all'ultimo termine ottenuto.

1.3 Lambda Calcolo e Sistemi dei Tipi

Consideriamo adesso il termine:

$$(\lambda x.\,\mathsf{not}\ x)\ \mathsf{not}$$

Un programma del genere non ci restituirà un risultato: stiamo applicando una funzione che opera su valori booleani ad un operatore logico. Eppure, quello risultante è un termine corretto secondo la nostra grammatica, ovvero è possibile costruire un programma fatto in questo modo. Per quanto riguarda

la sua valutazione invece, l'esecuzione si bloccherà dopo l'applicazione della regola di β-riduzione, in uno stato che non possiamo considerare un risultato accettabile per la computazione:

$$(\lambda x.\,\texttt{not}\ x)\ \texttt{not} \rightarrow_\beta \texttt{not}\ \texttt{not}$$

Ci piacerebbe che il nostro linguaggio non considerasse accettabili programmi che presentano evidenti incongruenze come quella vista prima, e per raggiungere questo scopo introdurremo ora il concetto di sistema di tipi, o type system.

Con type system intendiamo un insieme di regole sintattiche aventi lo scopo di assegnare un tipo ad alcuni elementi del linguaggio. Associare un tipo ad ogni termine ci permette di definire nuove regole per la costruzione dei termini, più raffinate di quelle date dalla grammatica iniziale del λ-calcolo, siccome tengono in considerazione anche i tipi dei termini e possono essere quindi più restrittive nel definire quali programmi si possano definire corretti e quali invece non lo siano.

Per prima cosa, è necessario definire attraverso una grammatica la struttura dei tipi che useremo nei nostri programmi:

$$\text{Type} := \text{Bool} \mid \text{Type} \rightarrow \text{Type}$$

Bool : Il tipo associato ai valori booleani.

Type $\rightarrow$ **Type** : Rappresenta il tipo delle funzioni. Il caso più semplice è rappresentato da Bool $\rightarrow$ Bool, ossia dal tipo di una funzione che accetta come parametro un booleano e che restituisce un booleano come risultato. Possiamo però avere anche casi più complessi: il tipo Bool $\rightarrow$ Bool $\rightarrow$ Bool è associato ad una funzione che prende come parametro un booleano e che restituisce a sua volta una funzione che prende come parametro un nuovo booleano e che restituisce un booleano come risultato.

Una volta definiti i nostri tipi, possiamo includerli nel nostro linguaggio attraverso una piccola modifica nella sintassi dei nostri programmi:

$$\lambda x\,.\,\mathsf{not}\ x\ :\ \mathsf{Bool}\ \rightarrow\ \mathsf{Bool}$$

Ogni termine viene ora seguito dal suo tipo.

Gli environment: Con l'introduzione di un sistema di tipi, vogliamo inoltre aggiungere ad un termine informazioni riguardo possibili assunzioni sui tipi delle variabili che vi compaiono. Definiamo l'environment di un termine come una lista di associazioni nella forma $\mathsf{Nome\ della\ variabile\ :\ Tipo\ della\ variabile}$ che rappresentano le nostre assunzioni. Vediamo ora un esempio:

$$\mathsf{x : Bool\,,\ y : Bool}\ \vdash\ \mathsf{and\ x\ y\ :\ Bool}$$

Alla destra del simbolo $\vdash$ abbiamo un normale programma del lambda calcolo con il relativo tipo, mentre sulla sinistra troviamo il suo environment, che nell'esempio presenta due associazioni che esprimono la conoscenza del tipo delle due variabili che compaiono nel termine.

Possiamo ora presentare le regole che costituiscono il nucleo del nostro type system, e che permettono, a seconda dei punti di vista, di assegnare un tipo ad ogni termine o di definire i passi corretti nella costruzione di un termine seguendo una logica più vincolata. Iniziamo con i primi assiomi del nostro sistema di tipi[1]:

$$\frac{}{\Gamma \vdash \mathrm{tt} : \mathrm{Bool}}\ (\mathrm{tt}) \qquad \frac{}{\Gamma \vdash \mathrm{ff} : \mathrm{Bool}}\ (\mathrm{ff})$$

$$\frac{}{\Gamma \vdash \mathrm{not} : \mathrm{Bool} \rightarrow \mathrm{Bool}}\ (\mathrm{not})$$

$$\frac{}{\Gamma \vdash \mathrm{and} : \mathrm{Bool} \rightarrow \mathrm{Bool} \rightarrow \mathrm{Bool}}\ (\mathrm{and})$$

$$\frac{}{\Gamma \vdash \mathrm{or} : \mathrm{Bool} \rightarrow \mathrm{Bool} \rightarrow \mathrm{Bool}}\ (\mathrm{or})$$

[1]Con i simboli Γ e Δ indicheremo insiemi, anche vuoti, di associazioni Variabile : Tipo

Queste regole hanno lo scopo di assegnare un tipo agli elementi base del nostro linguaggio. Vediamo ora alcune regole più interessanti:

$$\text{Variabile:} \quad \frac{}{\Gamma, \mathrm{x} : \pi \vdash \mathrm{x} : \pi}\ (\text{Var})$$

Questa regola dice semplicemente che se abbiamo come ipotesi che la variabile di nome x abbia tipo π, allora la variabile x ha tipo π.

$$\text{Applicazione:} \quad \frac{\Gamma \vdash \mathrm{M} : \pi \to \sigma \qquad \Delta \vdash \mathrm{N} : \pi}{\Gamma, \Delta \vdash \mathrm{M\ N} : \sigma}\ (\text{App})$$

L'applicazione di funzione tiene ora conto anche del tipo del parametro che la funzione accetta, in modo tale da impedirne usi impropri. Osservare come Γ e Δ siano insiemi distinti.

$$\text{Astrazione:} \quad \frac{\Gamma, \mathrm{x} : \pi \vdash \mathrm{M} : \sigma}{\Gamma \vdash \lambda \mathrm{x.M} : \pi \to \sigma}\ (\text{Abs})$$

La regola di astrazione è importante perchè stabilisce un collegamento tra i parametri delle funzioni e l'environment del termine. Attraverso questa è possibile, in un'ottica bottom-up, ridurre la complessità del termine in esame aggiungendo al contempo una nuova ipotesi al nostro ambiente.

Grazie alle regole sopra esposte, la costruzione dei termini del lambda calcolo è ora maggiormente vincolata, permettendo la riduzione di un gran numero di errori che si sarebbero potuti presentare e tempo di esecuzione. Ricordiamo comunque che esistono tipologie di errori che non possono essere prevenuti neanche utilizzando un efficiente type system.

1.4 L'Aggiunta del Probabilismo

La versione del lambda calcolo che utilizzeremo è stata ulteriormente arricchita con l'aggiunta delle costanti booleane probabilistiche: una costante di questo tipo esprime un dato booleano di cui non conosciamo se il valore che assume sia vero o falso, ma di cui abbiamo soltanto un'informazione sulla probabilità che valga `tt` oppure `ff`. Riguardo quest'ultimo punto, il nostro linguaggio attribuirà ad ognuna di queste costanti una probabilità del 50% di assumere una valore piuttosto che un altro. Sintatticamente, le costanti probabilistiche verranno introdotte con una costruzione come questa,

```
ProbabilityAtom
```

Ogni atomo probabilistico è indipendente dagli altri, ed il suo valore viene determinato solo in fase di valutazione del programma. L'introduzione del probabilismo ha un profondo impatto sull'intera semantica dei nostri programmi, ed in particolare sulla concezione del risultato di una computazione. L'esito della valutazione di un termine non è più un semplice valore booleano, ma consiste in una assegnazione di valori di probabilità ai suoi possibili risultati. Vediamo un semplice esempio:

```
and ProbabilityAtom ProbabilityAtom
```

Il risultato dalla valutazione di questo termine dipenderà dal valore assunto dagli atomi probabilistici che vi compaiono: l'output del programma varrà `tt` con una probabilità pari a 0.25, corrispondente alla probabilità che entrambe le costanti assumano valore `tt`.

Per concludere, aggiungiamo una nuova regola al nostro sistema di tipi per includere anche le costanti probabilistiche:

$$\frac{}{\Gamma \vdash \text{ProbabilityAtom} : \text{Bool}}\ (\text{prob})$$

Il tipo assegnato è chiaramente booleano, siccome quello è il tipo posseduto dai possibili valori che la costante può assumere.

Capitolo 2

ProbLog

Presenteremo ora il linguaggio di destinazione per il nostro traduttore. Inizieremo con l'introdurre brevemente l'interessante paradigma a cui fa riferimento, che si discosta nettamente da quello affrontato nel Capitolo 1 e dagli usuali paradigmi orientati agli oggetti. Vedremo dunque Prolog, uno dei maggiori esponenti di questa filosofia di programmazione, e successivamente ProbLog, una sua evoluzione nata con lo scopo di arricchire il linguaggio con elementi probabilistici. Per approfondimenti riguardo Prolog ed il paradigma di programmazione logica, rimando alla lettura di [2], mentre per chi volesse intraprendere lo studio del linguaggio ProbLog, le risorse disponibili sono consultabili a partire da [3].

2.1 Il Paradigma Logico

I vari paradigmi di programmazione si differenziano nel modo in cui scelgono di modellare le realtà di interesse, al fine di essere validi strumenti per la risoluzione di problemi. Il paradigma di programmazione logica rappresenta la realtà sotto forma di teorie logiche, ossia sistemi formali deduttivi in cui da premesse assunte vere è possibile derivare nuove conclusioni attraverso quella che definiamo essere un'elaborazione. Un programma diventa quindi un semplice insieme di proposizioni e di relazioni logiche definite su di esse: l'enfasi

non viene più posta sul flusso di esecuzione, ma sulla descrizione in termini logici del contesto del problema, lasciando che sia l'interprete del linguaggio ad elaborare le informazioni contenute nel programma per formulare le risposte ai nostri quesiti. Siccome un programma non viene più visto come una sequenza di operazioni o funzioni da valutare in un preciso ordine, diventa lecito domandarsi in cosa consista la valutazione di un programma scritto in un linguaggio affine a questo paradigma. Una volta definita la nostra teoria, è possibile sfruttarla attraverso il meccanismo delle interrogazioni, ossia domande poste al nostro modello riguardanti la dimostrabilità di determinate proposizioni.

2.2 Prolog

Prolog è uno dei principali linguaggi che implementa il paradigma di programmazione logica, nonchè la base per lo sviluppo di altri linguaggi che ne estendono le potenzialità. Un programma Prolog è costituito da un insieme di clausole, di cui se ne distinguono due tipologie: i **fatti** e le **regole**.

2.2.1 I Fatti

Un fatto rappresenta un assioma della nostra teoria, ossia una proposizione assunta vera per definizione:

```
piove.
```

Quello scritto sopra è un fatto, e costituisce già da solo un programma Prolog. Come già detto in precedenza, è possibile utilizzare un programma logico attraverso il meccanismo delle interrogazioni:

```
?- piove.
```

Quella sopra rappresenta la sintassi usata da Prolog per definire le interrogazioni: si fa seguire a `?-` il termine che ci chiediamo essere dimostrabile (o meno) nel nostro modello. Nell'esempio appena esposto, stiamo chiedendo

al programma se sia possibile derivare il termine piove. Eseguendo l'interrogazione sul programma riportato prima, la risposta che otterremo sarà yes. Se proviamo invece ad eseguire:

```
?- sole.
```

Prolog ci risponderà con no, siccome non è riuscito a dimostrare il termine sole a partire dagli assiomi che gli avevamo fornito. Fondamentalmente, quello che l'interprete Prolog ha cercato di fare è il matching sintattico tra il termine che gli abbiamo fornito nell'interrogazione e i termini definiti nel nostro programma. Nel primo caso abbiamo avuto un match, nel secondo no.

Quello visto prima apparteneva alla tipologia più semplice di fatti, rappresentati da nomi che Prolog vede come identificatori. I fatti di questo tipo vengono chiamati anche atomi. Un primo modo che Prolog mette a disposizione per creare relazioni tra proposizioni, è rappresentato dall'utilizzo dei termini composti, ossia nella forma:

$$\texttt{identificatore}(\texttt{Termine}_1, \ldots, \texttt{Termine}_n).$$

Un termine composto è un fatto che incorpora al suo interno altri termini (composti o meno), e che stabilisce una relazione tra di essi. Vediamo ora un programma che utilizza questo nuovo costrutto:

```
alberto.
matteo.
francesca.
uomo(alberto).
uomo(matteo).
donna(francesca).
amici(alberto, matteo).
amici(alberto, francesca).
```

Nell'esempio, uomo stabilisce una relazione che include alberto e matteo, ma non francesca.

Come facciamo se vogliamo sapere l'insieme dei termini che appartengono ad una data relazione? Possiamo utilizzare nelle nostre interrogazioni lo strumento che Prolog ci mette a disposizione: le variabili.

```
?-uomo(X).
```

Sintatticamente le variabili sono caratterizzate dalla lettera iniziale maiuscola, e poste all'interno di un'interrogazione richiedono all'interprete l'insieme dei termini che sostituiti alla variabile rendono vero il termine oggetto della domanda. Nell'esempio sopra esposto, X è una variabile, e l'interprete a seguito dell'interrogazione ci restituisce il seguente risultato:

```
X = alberto
X = matteo
```

che ci indica come i termini alberto e matteo sostituiti ad X costituiscano la risposta alla nostra domanda.

Attraverso il meccanismo delle variabili, è possibile definire interrogazioni di maggiore complessità:

```
?-amici(alberto, X), donna(X).
```

In Prolog la virgola ha valenza di AND per le nostre proposizioni, quindi quello che stiamo chiedendo adesso è l'insieme dei termini che rendano dimostrabile sia amici(alberto, X) che donna(X). La risposta dell'interprete in questo caso sarà semplicemente:

```
X = francesca
```

2.2.2 Le Regole

La seconda tipologia di clausola presente in Prolog, è rappresentata dalle regole. Una regola in Prolog possiede la seguente forma:

```
testa :- corpo
```

e viene interpretata come l'implicazione logica:

```
testa ⇐ corpo
```

Dove la testa della regola è un singolo fatto, mentre il corpo è costituito da una sequenza di uno o più fatti messi in relazione da congiunzioni e disgiunzioni logiche. Vediamo un esempio:

```
studia(marco).
sostiene_esami(marco).
buoni_voti(marco) :- studia(marco), sostiene_esami(marco).
```

Se marco studia e sostiene gli esami, allora prenderà buoni voti. Questo significa anche che se vogliamo provare a dimostrare che marco prende buoni voti, possiamo ricondurci a dimostrare che marco studia e che sostiene gli esami. Quando chiediamo all'interprete di dimostrare il fatto:

```
?- buoni_voti(marco)
```

per prima cosa, scorre la lista di fatti e regole che abbiamo definito nel nostro programma alla ricerca di un fatto o della testa di una regola che sia uguale al termine dell'interrogazione. Se il risultato della ricerca è un fatto, l'interprete può già terminare con successo la dimostrazione. Se invece quella trovata è una regola, l'interprete proverà a dimostrarne il corpo, che nel nostro esempio consiste nelle due proposizioni studia(marco) e sostiene_esami(marco). Siccome possiamo avere più regole con la stessa testa, è possibile che alcune di queste abbiano un corpo dimostrabile nel nostro modello ed altre no. Non approfondiremo troppo l'argomento, ma per gestire i più percorsi possibili che portano alla dimostrazione di un termine, l'interprete Prolog fa uso di un meccanismo di backtracking, che gli permette di tornare su i propri passi nel corso di una dimostrazione in caso quest'ultima non riesca a concludersi.

Nella scorsa sezione, abbiamo osservato come le variabili rappresentino un potente strumento utilizzabile nelle nostre interrogazioni. Vedremo adesso come queste possano essere utilizzate anche nella definizione delle regole per renderle più generali. Riprendiamo la nostra regola precedente:

```
buoni_voti(marco) :- studia(marco), sostiene_esami(marco).
```

questa è stata definita soltanto per marco, quindi un'interrogazione come:

```
?- buoni_voti(matteo)
```

otterrà una risposta negativa. Eppure, sarebbe logico pensare che qualunque studente se studia e sostiene gli esami, prenderà buoni voti. Utilizzando le variabili, è possibile generalizzare la nostra regola nel modo seguente:

```
buoni_voti(X) :- studia(X), sostiene_esami(X).
```

Alla variabile X è ora possibile sostituire qualunque termine, rendendo quindi possibile la dimostrazione di un fatto come buoni_voti(matteo), a patto ovviamente che nel programma siano presenti le proposizioni indicanti che matteo studia e sostiene gli esami.

2.3 ProbLog: Logica Probabilistica

Finora i programmi logici che abbiamo trattato erano stati concepiti con lo scopo di modellare situazioni in cui *tutto* potesse essere classificato come vero o come falso in senso assoluto. Gli assiomi definiti da noi erano considerati sempre veri, cosi come tutto ciò che a partire da essi poteva essere dedotto. Abbiamo quindi lavorato con un concetto "statico" di verità. ProbLog è un linguaggio che estende Prolog permettendo la possibilità di rappresentare realtà in cui determinati fatti o regole possano essere veri con una prefissata probabilità, ma in cui esiste anche l'ipotesi che non lo siano. ProbLog introduce infatti la possibilità di etichettare ogni termine con un valore di probabilità che indichi quanto è probabile che quella proposizione sia vera:

```
0.2::piove.
```

Se effettuiamo un'interrogazione all'interprete ProbLog riguardo la verità del fatto piove, questo non ci risponderà più con yes o no, ma ci dirà che la probabilità che il termine sia vero è del 20%.

Oltre ai fatti, anche le regole possono essere etichettate con una probabilità:

```
0.2::piove.
0.4::freddo.
0.3::nevica :- piove, freddo.
```

Il significato di questa regola è che assumendo che stia piovendo e faccia freddo, c'è una probabilità del 30% che nevichi. Se chiediamo all'interprete di calcolare la probabilità che nevichi, questa verrà calcolata in funzione delle probabilità dei singoli eventi che ne sono la causa, e nel nostro caso sarà uguale al prodotto

$$P(nevica) \;=\; P(piove) \cdot P(freddo) \cdot 0.3 \;=\; 0.024$$

2.4 La Semantica di ProbLog

A partire da un programma ProbLog, è possibile definire un insieme di mondi possibili caratterizzati dal verificarsi o no dei fatti che abbiamo specificato. Se il nostro programma comprende n fatti probabilistici, allora i mondi possibili saranno 2^n. In un mondo magari il primo fatto che abbiamo definito sarà falso, mentre il secondo sarà vero. In un altro mondo può valere il viceversa.

Una nota riguardo le regole: In ProbLog è possibile specificare sia fatti che regole con un'assegnata probabilità, ma in realtà la semantica del linguaggio prevede che solo i fatti possano essere probabilistici, mentre le regole siano da assumere sempre vere. Formalmente, è possibile adattarsi a questa nuova prospettiva tramite l'introduzione di un nuovo fatto probabilistico per ogni regola probabilistica definita:

```
0.2::piove.
0.4::freddo.
0.3::nevica :- piove, freddo.
```

diventa:

```
0.3::prob_neve.
0.2::piove.
0.4::freddo.
nevica :- piove, freddo, prob_neve.
```

È facile verificare come la probabilità di nevica non sia cambiata.

Una proposizione può risultare vera in un mondo e non esserlo in un altro, in base a se gli assiomi necessari per la sua dimostrazione siano presenti in quel mondo oppure no. Fissato l'insieme F dei fatti probabilistici definiti nel nostro programma, aventi la forma $p :: f$, ogni mondo possibile viene ora rappresentato da un insieme $F' \subseteq F$ contenente i soli fatti che in quel mondo sono verificati. Otteniamo quindi la seguente distribuzione di probabilità sull'insieme dei mondi possibili:

$$P_F(F') = \prod_{f_i \in F'} p_i \cdot \prod_{f_i \in F \setminus F'} (1 - p_i)$$

Possiamo ora definire la probabilità associata ad un'interrogazione q come la somma delle probabilità dei mondi in cui è possibile dimostrare q. Ovviamente, visto il numero esponenziale di mondi possibili associabili ad un programma ProbLog, questa definizione è puramente teorica ed utilizzata esclusivamente nella semantica del linguaggio. L'interprete ProbLog utilizza algoritmi differenti, ma che in questa sede non tratteremo. Al lettore interessato all'approfondimento della semantica di ProbLog, è consigliata la lettura di [4]. Infine, è interessante notare come ad ogni mondo possibile corrisponda un programma Prolog, siccome in ognuno di essi ogni fatto è univocamente definito come vero o come falso.

Capitolo 3

La Geometria dell'Interazione

La Geometria dell'Interazione è una struttura semantica per la logica lineare inventata dal logico Jean-Yves Girard [6]. Intuitivamente può sembrare difficile immaginare come questa possa aver trovato posto nello sviluppo del traduttore, ma vedremo come in realtà un programma del lambda calcolo sia intrinsecamente connesso a questo tipo di logica. Inizieremo il capitolo approfondendo tale collegamento, per poi studiare come questo sia stato effettivamente sfruttato per effettuare la traduzione tra i nostri due linguaggi. La Geometria dell'Interazione rappresenta un argomento complesso, ed in questa tesi, pur traendo ispirazione da tale semantica, si è scelto di presentarla in una versione più semplificata ed intuitiva, che non richiede particolari premesse.

3.1 Reinterpretare un Programma

Osservando la costruzione di un termine nel sistema formale del λ-calcolo tipato, vale la pena soffermarsi sulla struttura che assume il suo albero di derivazione:

$$\dfrac{\dfrac{\dfrac{}{\vdash \text{not} : \text{Bool} \to \text{Bool}}\,(\text{not}) \quad \dfrac{}{\text{x} : \text{Bool} \vdash \text{x} : \text{Bool}}\,(\text{Var})}{\text{x} : \text{Bool} \vdash \text{not x} : \text{Bool}}\,(\text{App})}{\vdash \lambda\text{x.not x} : \text{Bool} \to \text{Bool}}\,(\text{Abs}) \quad \dfrac{}{\vdash \text{tt} : \text{Bool}}\,(\text{tt})}{\vdash (\lambda\text{x.not x})\ \text{tt} : \text{Bool}}\,(\text{App})$$

La chiave del collegamento tra il nostro linguaggio e la Geometria dell'Interazione, risiede nel vedere l'albero di derivazione di un lambda-termine come una dimostrazione nella logica lineare. Per fare un esempio, osserviamo come la regola di applicazione del lambda calcolo tipato possegga la stessa struttura della regola logica di Modus ponens:

$$\dfrac{\Gamma \vdash \text{M} : \pi \to \sigma \qquad \Delta \vdash \text{N} : \pi}{\Gamma, \Delta \vdash \text{M N} : \sigma}\,(\text{App}) \quad \dfrac{\text{A} \Rightarrow \text{B} \qquad \text{A}}{\text{B}}\,(\text{Modus})$$

Se ci concentriamo sui soli tipi che compaiono nei termini della regola di applicazione, notiamo infatti come le due regole si equivalgano. Per capire il ruolo degli envirnoment presenti nei nostri termini, possiamo invece guardare alla regola di astrazione:

$$\dfrac{\Gamma, \text{x} : \pi \vdash \text{M} : \sigma}{\Gamma \vdash \lambda\text{x.M} : \pi \to \sigma}\,(\text{Abs}) \quad \dfrac{\begin{array}{c}[\text{A}] \\ \vdots \\ \text{B}\end{array}}{\text{A} \Rightarrow \text{B}}\,(\Rightarrow_{\text{i}})$$

Le variabili presenti negli envirnoment rappresentano assunzioni. Guardando la regola di astrazione dal basso verso l'alto, quello che facciamo consiste nell'assumere di avere una variabile x di tipo π, siccome questa ci viene fornita dall'input della funzione sotto.

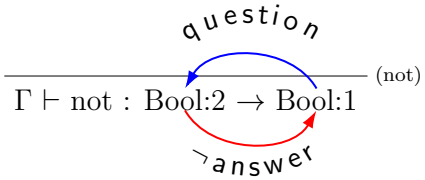
Abbiamo quindi che la costruzione di un programma corrisponde alla dimostrazione di una proposizione logica, e proprio grazie all'esistenza di questo parallelismo, ci sarà possibile sfruttare un importante risultato come la Geometria dell'Interazione nello sviluppo del nostro traduttore.

La Geometria dell'Interazione ci permette di vedere i lambda-termini, o programmi, sotto un nuovo punto di vista: nel corso della derivazione, ogni termine intermedio può essere visto come un componente, o un circuito se preferiamo, che viene combinato con altri per costruire il termine finale, ossia il nostro programma. In quest'ottica, i tipi dei termini rivestono un ruolo di fondamentale importanza: rappresentano le interfacce di collegamento utilizzate per mettere in comunicazione i vari componenti del nostro sistema. Nel dettaglio, i nostri componenti comunicano scambiando domande e risposte con gli altri termini a cui sono collegati: un componente può domandare il valore assunto dai dati dei tipi che prende in input, mentre deve essere in grado di rispondere ad eventuali domande sul valore dei suoi output.

3.2 Le Regole di Inferenza Secondo La Nuova Semantica

Per definire con chiarezza come la Geometria dell'Interazione modelli la nostra concezione dei programmi del lambda calcolo, è opportuno vedere come questa interpreta le varie regole utilizzate nella costruzione dei termini del linguaggio. Iniziamo con la regola associata all'operatore logico NOT:

Operatore NOT:



La regola del not, i cui tipi sono stati numerati per semplificarne il riferimento, è un componente che presenta un input sul tipo Bool:2, ossia il parametro

della funzione not, e che fornisce come output nel tipo Bool:1 il proprio input negato. Le frecce in figura mostrano un primo esempio di comunicazione tra tipi, in questo caso appartenenti allo stesso termine. Quando viene ricevuta un'interrogazione sul valore assunto da Bool:1, internamente il componente genera un'ulteriore domanda riguardante questa volta il valore assunto da Bool:2. Una volta che questa avrà avuto una risposta, il valore ottenuto su Bool:2 verrà negato e inviato come risposta alla domanda iniziale riguardante il valore di Bool:1.

Proviamo ora a tradurre le precedenti interazioni in codice logico: i tipi presenti nei termini sono collegati tra loro attraverso due diverse tipologie di collegamenti, le domande e le risposte. Una domanda dal tipo con indice i a quello con indice j, in ProbLog assumerà questa forma:

```
q(j) :- q(i).
```

La lettera q deriva da "question". Il significato di questa semplice regola logica è che una domanda riguardante il valore del tipo i, implicherà l'esistenza di una domanda riguardante questa volta il valore del tipo j. Una risposta dal tipo j verso il tipo i si presenterà invece in questo modo:

```
ans(B, i) :- ans(B, j).
```

La regola fa uso della variabile B, che conterrà il valore (nel nostro caso per forza booleano) che verrà fornito come risposta. Formalmente, questa regola ci dice che se la risposta alla domanda sul valore di j è B, allora la risposta alla domanda sul valore di i sarà B.

Seguendo questa sintassi, la regola associata all'operatore NOT si tradurrà quindi nel seguente codice logico:

```
q(2) :- q(1).
ans(X,1) :- ans(B,2), not(B,X).
```

Attraverso la prima clausola specifichiamo che una domanda riguardante il valore del tipo Bool:1 deve generare una domanda per il tipo Bool:2. Giustamente affinchè la funzione NOT possa restituire un risultato, è necessario che

questa conosca il valore del proprio input. A questo punto, la seconda regola definisce che: sia B il valore associato al tipo Bool:2, rappresentante l'input della funzione NOT, allora il valore associato al tipo Bool:1 sarà X, dove X è la negazione di B.

Costanti Booleane: $\dfrac{}{\Gamma \vdash \mathrm{tt} : \mathrm{Bool{:}i}}\ (\mathrm{tt})$ $\dfrac{}{\Gamma \vdash \mathrm{ff} : \mathrm{Bool{:}i}}\ (\mathrm{ff})$

Prendiamo la regola tt (per la regola ff il discorso è analogo): il tipo Bool:i rappresenta l'unico output che il componente espone, mentre non necessita di alcun input, se non quelli provenienti dal suo environment. Le variabili contenute nell'environment sono infatti da considerarsi input per il termine, siccome queste rappresentano valori che possiamo assumere di avere, o per meglio dire, di ricevere dall'esterno. Idealmente, questi due assiomi possono essere visti come generatori dei rispettivi valori. La regola tt ad esempio, produce il valore tt, e nel farlo non richiede particolari input. Il codice che associeremo agli assiomi sopra citati sarà dunque

```
ans(tt,i) :- q(i).
```

per la regola tt, oppure

```
ans(ff,i) :- q(i).
```

nel caso della regola ff. Il significato di queste clausole è che una domanda riferita ad un componente tt o ff genera come risposta un'associazione del valore tt/ff al tipo oggetto della domanda.

Atomo Probabilistico: $\dfrac{}{\Gamma \vdash \mathrm{ProbabilityAtom} : \mathrm{Bool{:}i}}\ (\mathrm{prob})$

Per quanto riguarda invece la regola associata alle costanti probabilistiche, anche lei rappresenta un generatore di valori booleani, con la differenza rispetto alle due regole precedenti, che questa volta il valore generato non è determinato. Per esprimere in ProbLog un atomo probabilistico è possibile utilizzare un fatto nella forma:

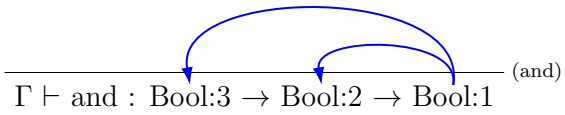
```
0.5::probatom(i).
```

dove i è l'indice associato al tipo della costante. Potremo quindi completare la nostra traduzione con il seguente codice logico:

```
ans(tt,i) :- q(i), probatom(i).
ans(ff,i) :- q(i), \+probatom(i).
```

In ProbLog, fissato un fatto f, indichiamo con $\backslash+f$ il non verificarsi di f. La prima delle due clausole ci dice che se abbiamo una domanda per il tipo i ed il fatto probatom(i) è verificato, allora assoceremo ad i la risposta tt, ossia la nostra costante probabilistica varrà True. Se invece sarà \+probatom(i) a verificarsi, varrà la seconda clausola, che assocerà alla costante il valore ff. Siccome probatom(i) sarà verificato o meno con probabilità pari a 0.5, questa sarà anche la probabilità che una domanda al tipo i restituisca valore tt piuttosto che ff.

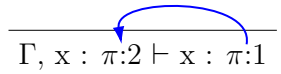
Operatore AND:



Le regole and e or sono molto simili, quindi analizzeremo soltanto la prima. In questo caso, a seguito di una domanda sul valore assunto da Bool:1, vengono generate due diverse interrogazioni sui tipi Bool:2 e Bool:3. Una volta che il componente and avrà ricevuto entrambe le risposte alle sue domande, potrà procedere con l'effettuare l'operazione di AND tra i due valori ottenuti, usando il risultato come risposta alla domanda iniziale sul tipo Bool:1.

```
q(2) :- q(1).
q(3) :- q(1).
ans(X,1):-ans(A,2), ans(B,3), and(A,B,X).
```

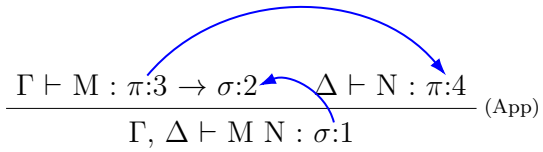
In modo simile a quanto fatto per l'operatore NOT, il valore associato a Bool:1 sarà X = A AND B , dove A e B sono i valori assunti dai due tipi Bool:2 e Bool:3. La regola associata alla funzione OR viene tradotta in modo analogo, con l'unica differenza che al posto di and(A,B,X) troviamo or(A,B,X).

Variabile: 

La regola sopra mette in atto una cosiddetta tecnica di copycat per gestire le domande riguardanti il suo output: come possiamo vedere, la regola presenta un input ed un output aventi lo stesso tipo e la stessa etichetta x, il che implica che la risposta alla domanda su uno dei due tipi, sia una risposta valida anche per l'altro. Quando viene ricevuta una domanda sul tipo π:1, questa viene girata direttamente al tipo π:2, che è un input per la regola. Successivamente, la risposta ottenuta verrà inoltrata al tipo π:1, senza modificarla in alcun modo.

```
q(2) :- q(1).
ans(X,1) :- ans(X,2).
```

La traduzione di questa regola presenta però un problema: il codice sopra riportato è corretto nel caso in cui il tipo π consista in un semplice booleano, ma non va bene se con π identifichiamo un tipo funzionale, di natura quindi più complessa. Nel Capitolo 4 studieremo come creare collegamenti tra tipi funzionali, ma per ora ci limiteremo ad assumere che i tipi considerati siano tutti Bool.

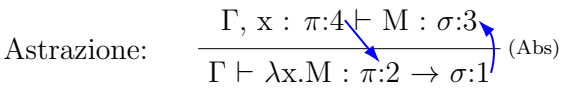
Applicazione: 

La regola di applicazione ci permette ci collegare fra loro componenti diversi per costruirne uno nuovo. Le frecce in figura mostrano la direzione delle interrogazioni, ma ognuna di esse è da intendersi come un copycat. In questo contesto abbiamo due diverse linee di collegamento (bidirezionali): La prima, che va da σ:1 a σ:2, ha lo scopo di inoltrare un'eventuale domanda sul primo tipo al secondo. Il valore prodotto dal termine M N in σ:1 sarà infatti quello prodotto in output dalla funzione M, applicata al termine N. Lo scopo del secondo collegamento è proprio quello di permettere alla funzione M di richiedere il valore del proprio input π:3 al termine N, che espone il tipo

π:4 come interfaccia per il suo output. Questa volta associeremo alla nostra regola il seguente codice,

```
q(2) :- q(1).
ans(X,1) :- ans(X,2).

q(4) :- q(3).
ans(X,3) :- ans(X,4).
```

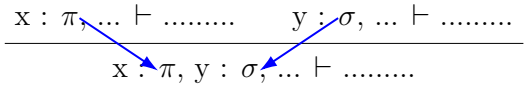
sotto l'ipotesi che i tipi π e σ siano Bool:



Anche la regola di astrazione introduce due nuovi collegamenti: il primo, che connette π:4 a π:2, permette alla variabile x di essere spostata nell'environment del termine soprastante senza perdere il riferimento a dove questa era stata definita. Il secondo collegamento invece, indica che una domanda riguardante il valore di σ:1 dovrà essere inoltrata al tipo σ:3, siccome in entrambi i casi ci stiamo riferendo al valore assunto dal termine M. In termini logici l'astrazione verrà espressa come:

```
q(3) :- q(1).
ans(X,1) :- ans(X,3).

q(2) :- q(4).
ans(X,4) :- ans(X,2).
```

Concludiamo con un'ultima regola di carattere generale, che va integrata a tutte le regole finora analizzate:

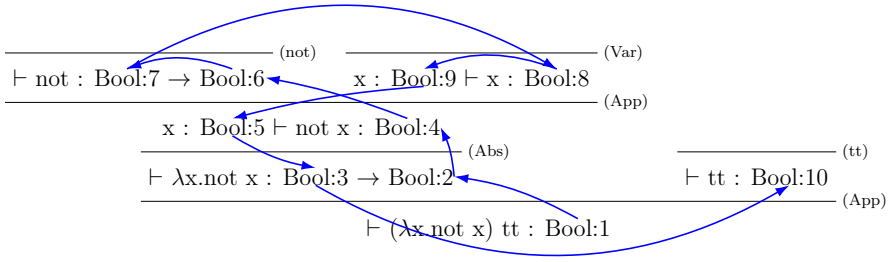


Durante la derivazione, partendo dai componenti che assumiamo di avere e che costituiscono le premesse delle nostre regole, costruiamo nuovi componenti che rappresentano le conclusioni. Se vediamo l'insieme delle variabili

nell'environment di un termine come un insieme di input per il componente associato a tale termine, diventa chiara la necessità di introdurre dei collegamenti che permettano agli input dei componenti interni, ossia quelli nelle premesse, di interfacciarsi con gli input dei componenti esterni, ovvero gli input delle conclusioni. Per questa ragione, ogni variabile presente nell'environment di una premessa viene collegata alla corrispettiva variabile nell'environment della conclusione.

3.3 Un Programma Come un Flusso di Dati

Analizziamo adesso un intero programma attraverso la nuova semantica che abbiamo introdotto:



Anche questa volta sono state indicate in figura soltanto le frecce raffiguranti le domande scambiate fra i diversi tipi, ma non bisogna dimenticarsi che lo schema si compone anche delle risposte, come illustrato nella sezione precedente. Come possiamo notare, quello che viene a generarsi è un flusso di dati, sotto forma di domande e risposte, che di fatto costituisce la computazione. Per avviarla, si pone una domanda iniziale sul valore assunto dal tipo Bool:1, rappresentante l'output del nostro programma, e a seguito dell'attraversamento dell'albero di derivazione, la computazione terminerà fornendo una risposta sul valore di Bool:1.

La Geometria dell'Interazione ci fornisce quindi una visione totalmente diversa del nostro programma, costituita da un insieme di collegamenti che descrivono come i singoli termini interagiscano fra loro. Una rappresentazione del genere semplifica enormemente il lavoro di traduzione del programma in

codice logico, siccome i singoli collegamenti che compongono il programma sono facilmente modellabili in ProbLog come fatti e regole logiche.

La scelta di utilizzare questa struttura semantica permette inoltre un approccio modulare al processo di traduzione. Prendiamo come esempio il programma

$$\dfrac{\dfrac{}{\vdash \text{not} : \text{Bool:3} \to \text{Bool:2}}\,(\text{not}) \qquad \dfrac{\dfrac{}{\vdash \text{not} : \text{Bool:6} \to \text{Bool:5}}\,(\text{not}) \qquad \dfrac{}{\vdash \text{tt} : \text{Bool:7}}\,(\text{tt})}{\vdash \text{not tt} : \text{Bool:4}}\,(\text{App})}{\vdash \text{not (not tt)} : \text{Bool:1}}\,(\text{App})$$

Partendo dall'ipotesi che i tipi che compaiono nell'albero siano tutti univocamente identificabili, è possibile spezzare in più parti il processo di traduzione, traducendo separatamente i vari sotto alberi che formano il nostro termine. Nel caso del nostro programma, possiamo iniziare col tradurre il suo sotto albero destro, che chiameremo π:

$$\pi: \quad \dfrac{\dfrac{}{\vdash \text{not} : \text{Bool:6} \to \text{Bool:5}}\,(\text{not}) \qquad \dfrac{}{\vdash \text{tt} : \text{Bool:7}}\,(\text{tt})}{\vdash \text{not tt} : \text{Bool:4}}\,(\text{App})$$

Denoteremo quindi con P_π il programma logico ad esso associato:

```
q(5):-q(4).
ans(B,4):-ans(B,5).
q(7):-q(6).
ans(B,6):-ans(B,7).
q(6):-q(5).
ans(X,5):-ans(B,6), not(B,X).
ans(tt,7):-q(7).
```

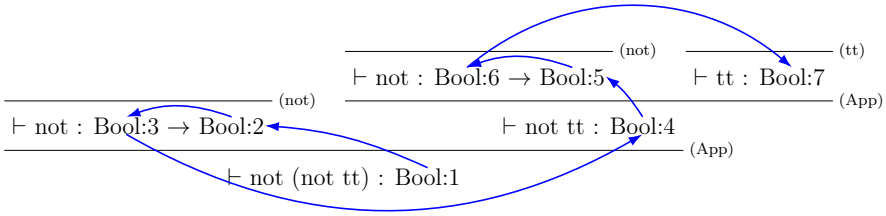
Possiamo ora effettuare la traduzione del sotto albero sinistro del nostro programma, rappresentante la funzione che vogliamo applicare al termine precedentemente tradotto:

$$\rho: \quad \frac{}{\vdash \text{not} : \text{Bool:3} \to \text{Bool:2}}\,(\text{not})$$

Il codice logico P_ρ prodotto sarà quindi

```
q(3):-q(2).
ans(X,2):-ans(B,3), not(B,X).
```

A questo punto è possibile utilizzare la regola di applicazione per collegare i due programmi, in modo tale da generare il risultato finale della traduzione:



Denotiamo con P_ν l'insieme delle clausole ProbLog che realizzano il collegamento tra i due termini:

```
q(2):-q(1).
ans(B,1):-ans(B,2).
q(4):-q(3).
ans(B,3):-ans(B,4).
```

Avremo quindi che il codice logico associato all'intero albero sarà dato da

$$P_\pi \cup P_\rho \cup P_\nu \cup \{...\}$$

dove oltre alle clausole precedentemente analizzate sarà necessario aggiungere un ridotto insieme di clausole ausiliarie, le quali non dipendono dal programma che andiamo a tradurre e che esamineremo in dettaglio nel prossimo capitolo.

Nell'esempio precedente abbiamo scelto di numerare i tipi compresi nell'albero prima di effettuare le traduzioni dei sotto alberi, in modo tale che ogni tipo avesse un proprio indice univoco. In determinate circostanze può diventare necessario effettuare un'applicazione tra due termini tradotti separatamente, i cui insiemi di clausole utilizzano numerazioni distinte che possono generare conflitti in fase di unione. In tal caso sarà necessario effettuare una reindicizzazione dei tipi che compaiono nelle varie regole logiche, in modo tale che gli insiemi risultanti possano essere combinati come visto in precedenza.

Capitolo 4

Implementazione Del Traduttore

In questo ultimo capitolo verrà affrontata l'effettiva realizzazione del traduttore. Il software è stato scritto nel linguaggio di programmazione Python, seguendo il paradigma orientato agli oggetti, e in tutte le fasi dello sviluppo è stata prestata particolare attenzione affinchè il codice risultasse il più possibile modulare, in modo tale da facilitarne eventuali estensioni future. La maggior parte degli algoritmi utilizzati è costruita per induzione strutturale sull'albero di derivazione del lambda-termine di cui viene effettuata la traduzione, che costituisce l'input del traduttore. Inizieremo col vedere il modo in cui le varie componenti dei linguaggi lambda calcolo e ProbLog sono state modellate nel codice Python, per poi studiare nel dettaglio come si articola il processo di traduzione.

4.1 Lambda-Termini in Python

I programmi in ingresso al traduttore sono espressi sotto forma di alberi di derivazione, costituiti da concatenazioni di regole di inferenza. La prima fase dello sviluppo ha riguardato quindi la modellizzazione degli alberi come og-

getti Python, risultanti dalla combinazione di più componenti. Innanzitutto sono stati definiti i tipi associati ai termini:

```
1  class Type(object):
2      def __init__(self):
3          pass
4
5  class Arrow(Type):
6      def __init__(self, left:Type, right:Type):
7          self.left = left
8          self.right = right
9
10 class SimpleType(Type):
11     def __init__(self):
12         self.index = None # Utilizzato nella
13                           # numerazione dei tipi
14
15 class Bool(SimpleType):
16     def __init__(self):
17         super().__init__()
```

Per realizzare un'architettura il più possibile estendibile, è stata sfruttata la tecnica dell'ereditarietà: tutti i tipi sono sottoclassi di Type, ma vengono distinti i tipi semplici, nel nostro caso rappresentati dal solo Bool, da quelli composti come Arrow, che individua il tipo funzionale. Ogni tipo semplice possiede inoltre un attributo index, che corrisponde al numero utilizzato per identificarlo univocamente all'interno dell'albero di derivazione.

Un termine del lambda calcolo è rappresentato attraverso la classe Expression, a sua volta composta da tre parti:

- L'**environment**, costituito da un dizionario contenente associazioni nella forma Nome_Variabile : Tipo.
- Un oggetto **Term** contenente la stringa che raffigura il termine.
- Il **Tipo** del termine.

Per modellare gli alberi associati ai termini è stata definita una classe InferenceRule, che presenta un numero di sottoclassi pari a quello delle regole di inferenza utilizzate nel nostro sistema dei tipi.

```
1  class InferenceRule(object):
2  """Classe nodo dell'albero"""
3  def __init__(self, conclusion:Expression):
4      self.conclusion = conclusion
5
6
7  class AbstractionRule(InferenceRule):
8  def __init__(self, conclusion:Expression,
9                premise_one:InferenceRule):
10     super().__init__(conclusion)
11     self.premise_one = premise_one
12
13 ......Altre regole......
```

Le regole di inferenza formano una struttura dati ad albero, con la radice presente nella conclusione dell'albero di derivazione. Ogni regola possiede una propria conclusione costituita da un'espressione, oltre che zero, una o più premesse che rappresentano i collegamenti verso le regole di inferenza superiori nella derivazione.

4.2 Rappresentazione dei Programmi Logici

La modellizzazione delle clausole che compongono i programmi ProbLog è risultata più semplice paragonata a quella richiesta per i lambda-termini. Come abbiamo visto nel Capitolo 3, la Geometria dell'Interazione ci permette di associare ad ogni regola di inferenza un insieme di collegamenti, che sono la base di partenza per la conversione del programma in codice logico. Le uniche clausole che è stato necessario implementare in Python sono quelle utilizzate per rappresentare questi collegamenti. Per rappresentare le interazioni tra i tipi, anche in questo caso è stata creata un sovra-classe GOIClause che viene estesa dalle varie classi che implementano ognuna un diverso tipo di istruzione logica.

```
1  class GOIClause(object):
2      def __init__(self):
3          pass
4
5  class GOIQImplication(GOIClause):
6      def __init__(self, from_index, to_index):
7          self.from_index = from_index
8          self.to_index = to_index
9
10     def __str__(self):
11         return
12           "q("+str(self.to_index)+"):-
13                         q("+str(self.from_index)+")."
14
15 class GOIAnsImplication(GOIClause):
16     def __init__(self, from_index, to_index):
17         self.from_index = from_index
18         self.to_index = to_index
19
20     def __str__(self):
```

```
21          return "ans(B,"+str(self.to_index)+"):-
22                              ans(B,"+str(self.from_index)+").
23
24  ......Altre clausole......
```

Ogni classe prevede la ridefinizione del metodo __str__(self), la cui stringa restituita sarà l'effettivo codice ProbLog associato alla clausola. Nel codice possiamo vedere le classi GOIQImplication e GOIAnsImplication, che estendono GOIClause e rappresentano rispettivamente un collegamento di tipo "question" e "answer".

4.3 Il Processo di Traduzione

Studieremo adesso le varie fasi che compongono il processo di traduzione. Il procedimento prevede di attraversare più volte l'albero di derivazione del termine preso in input, popolando al contempo l'insieme di clausole goi_clauses, che alla fine del processo costituirà il programma logico risultante dalla traduzione.

4.3.1 Definizione degli Operatori in ProbLog

Per prima cosa, inseriamo in goi_clauses le definizioni in termini logici degli operatori booleani utilizzati nei programmi del lambda calcolo.

```
not(tt, ff).
not(ff, tt).

and(tt, tt, tt).
and(tt, ff, ff).
and(ff, tt, ff).
and(ff, ff, ff).

or(tt, tt, tt).
```

```
or(tt, ff, tt).
or(ff, tt, tt).
or(ff, ff, ff).
```

I predicati cosi definiti potranno essere utilizzati nella traduzione delle corrispettive funzioni logiche nel lambda calcolo.

Oltre a queste, sono necessarie altre due clausole standard da aggiungere all'inizio di ogni operazione di traduzione:

```
q(1).
query(ans(B,1)).
```

L'algoritmo di indicizzazione dei tipi utilizzato nel traduttore assegna il numero 1 sempre al tipo del termine rappresentante la conclusione dell'albero di derivazione, il quale valore rappresenta quindi l'output del programma. La prima delle due clausole sopra citate serve proprio per porre la domanda iniziale su quale sia il valore assunto dal tipo numerato come 1. La seconda clausola invece, è il modo in cui si presentano le interrogazioni in ProbLog, e corrisponderebbe quindi ad:

```
?-ans(B,1).
```

ossia stiamo domandando all'interprete quali valori possa assumere la variabile B, rappresentante il valore di output del programma.

4.3.2 Numerazione dei Tipi

La prima operazione effettuata sul programma in ingresso consiste nella numerazione dei tipi presenti nel suo albero di derivazione. In questa fase vengono utilizzate due funzioni che attraversano l'albero associando ad ogni tipo semplice incontrato un intero crescente, che sarà in seguito utilizzato per riferirsi al tipo nella definizione dei collegamenti logici ProbLog.

```
1  def enumerate_type(term_type: Type, index):
2
3      if isinstance(term_type, SimpleType):
4          term_type.index = index
5          index += 1
6      elif isinstance(term_type, Arrow):
7          index = enumerate_type(term_type.right, index)
8          index = enumerate_type(term_type.left, index)
9
10     return index
```

La funzione enumerate_type accetta come parametri un tipo term_type più l'indice da cui far iniziare la numerazione, ed il suo compito consiste nel numerare tutti i tipi semplici che compaiono in term_type. La presenza del tipo Arrow permette ai tipi complessi di assumere una struttura ad albero binario, facilmente attraversabile attraverso l'utilizzo di funzioni definite per induzione strutturale. enumerate_type se invocata su un tipo semplice, assegna a quest'ultimo un indice, mentre nel caso in cui term_type sia di tipologia Arrow si richiama ricorsivamente sui due sotto alberi, facendo attenzione che gli indici assegnati proseguano con continuità senza ripetersi.

```
def enumerate_rule(inf_rule:InferenceRule, index):

    #Enumero il tipo della conclusione della regola
    index = enumerate_type(inf_rule.conclusion.term_type, index)

    # Enumero i tipi delle variabili contenute
    # nell'environment della conclusione
    for key, value in inf_rule.conclusion.env.var_dict.items():
        index = enumerate_type(value, index)

    if isinstance(inf_rule, ApplicationRule):
        index = enumerate_rule(inf_rule.premise_one, index)
        index = enumerate_rule(inf_rule.premise_two, index)
    elif isinstance(inf_rule, AbstractionRule):
        index = enumerate_rule(inf_rule.premise_one, index)

    return index
```

La funzione enumerate_rule fa uso della funzione enumerate_type per effettuare la numerazione dell'albero di derivazione associato al programma da tradurre. I parametri sono costituiti da una regola di inferenza rappresentante la radice dell'albero da attraversare, e dall'indice da cui far partire la numerazione. Anche in questo caso la funzione è definita per induzione su una struttura ad albero, ma sono presenti più casi che si distinguono in base al numero di premesse della regola attraversata. La funzione enumerate_type viene richiamata ogni volta che è necessario indicizzare un nuovo tipo, considerando anche quelli presenti negli environment dei termini.

4.3.3 Creazione delle Definizioni per le Costanti Probabilistiche

Successivamente alla numerazione dei tipi, l'albero viene attraversato una seconda volta per creare le definizioni ProbLog delle costanti probabilistiche che compaiono nel programma. Ogni qualvolta viene incontrata una regola di inferenza che introduce una nuova costante, si genera una clausola nella forma:

```
0.5::probatom(i).
```

dove i è il numero associato al tipo della costante probabilistica. Ogni atomo probabilistico presente nel programma funzionale viene mappato in un fatto differente all'interno del programma ProbLog, in modo tale che ogni costante sia indipendente dalle altre in fase di valutazione del programma logico. Ricordiamo infine che nell'operazione di traduzione si assume che tutte le costanti abbiano una probabilità associata pari a 0.5.

4.3.4 Creazione dei Collegamenti tra Tipi

Abbiamo visto come la Geometria dell'Interazione ci permetta di esprimere i nostri programmi nelle forma di componenti connesse attraverso le loro interfacce, rappresentate dai tipi che i loro termini espongono. Il collegamento tra tipi semplici è stato già affrontato, vedremo quindi ora come generalizzare il discorso precedente anche alla connessione tra tipi composti, ossia i tipi associati alle funzioni del lambda calcolo. Affinchè due tipi siano collegabili devono essere soddisfatte le seguenti condizioni:

1. I tipi devono possedere la stessa struttura, ossia devono rappresentare lo stesso tipo.

2. Nel contesto dell'ottica circuitale che abbiamo definito, sono consentiti soltanto collegamenti che interessano coppie di tipi in cui uno dei due costituisce un input per il relativo termine e l'altro un output.

Intuitivamente stiamo dicendo che le uscite si possono collegare soltanto alle entrate degli altri componenti e che entrate ed uscite devono possedere la stessa forma per potersi connettere. Siccome la distinzione tra tipi di input e di output per i termini è stata trattata nel Capitolo 3 in relazione alle regole di inferenza in cui compaiono, vedremo adesso come viene creato un collegamento tra due tipi funzionali. Prendiamo come esempio un'istanza della regola Variabile:

$$\frac{}{\text{x : Bool:6} \to \text{Bool:5} \to \text{Bool:4} \ \vdash \ \text{x : Bool:3} \to \text{Bool:2} \to \text{Bool:1}} \ (\text{Var})$$

Dobbiamo costruire un collegamento "question" che parta dal tipo del termine e che arrivi a quello nell'environment. Per farlo, introduciamo il concetto di polarità: come per i termini, anche all'interno di un singolo tipo composto è possibile distinguere alcuni tipi semplici che fungono da input ed altri che fungono da output. Riprendiamo il paragone visto nel terzo capitolo tra il tipo funzionale e l'implicazione logica:

$$\frac{\Gamma \vdash \text{M} : \pi \to \sigma \qquad \Delta \vdash \text{N} : \pi}{\Gamma, \Delta \vdash \text{M N} : \sigma} \ (\text{App}) \qquad \frac{\text{A} \Rightarrow \text{B} \qquad \text{A}}{\text{B}} \ (\text{Modus})$$

In entrambi i casi, abbiamo che sia il tipo π che la variabile proposizionale A, rappresentano una sorta di input nel loro contesto, mentre il tipo σ e la variabile B costituiscono output, ossia oggetti prodotti grazie alla presenza di un input al termine o alla formula logica. Questo parallelismo ci è particolarmente utile perchè la nostra distinzione tra ingressi ed uscite nei tipi composti trova una diretta corrispondenza nella negazione o meno delle variabili proposizionali che compongono una formula logica:

$$\text{A} \vdash \text{B} \ \equiv \ \text{A} \Rightarrow \text{B} \ \equiv \ \neg\text{A} \vee \text{B}$$

Abbiamo quindi che per determinare quali siano gli ingressi e le uscite in un tipo composto, è sufficiente verificare quali tipi semplici siano negati e quali no esprimendo il tipo composto come disgiunzione o congiunzione di variabili

proposizionali. Riprendendo il nostro esempio precedente, il tipo associato al termine corrisponde all'implicazione logica:

$$\text{Bool:3} \Rightarrow \text{Bool:2} \Rightarrow \text{Bool:1} \;\equiv\; \neg\text{Bool:3} \vee \neg\text{Bool:2} \vee \text{Bool:1}$$

da cui deduciamo che i tipi 2 e 3 siano ingressi per il termine x, mentre il tipo 1 sia un'uscita. Per quanto riguarda invece i tipi presenti nell'environment, il discorso è analogo ma dobbiamo ricordarci di negare le rispettive formule proposizionali associate:

$$\neg(\text{Bool:6} \Rightarrow \text{Bool:5} \Rightarrow \text{Bool:4}) \;\equiv\; \text{Bool:6} \wedge \text{Bool:5} \wedge \neg\text{Bool:4}$$

A questo punto è possibile tracciare i collegamenti tra i due tipi complessi, che nel caso delle domande andranno sempre dai tipi non negati, detti anche positivi, ai tipi negati o negativi.

$$\frac{}{\text{x} : \text{Bool:6} \to \text{Bool:5} \to \text{Bool:4} \;\vdash\; \text{x} : \text{Bool:3} \to \text{Bool:2} \to \text{Bool:1}}\ (\text{Var})$$

Le risposte seguiranno invece la direzione opposta:

$$\frac{}{\text{x} : \text{Bool:6} \to \text{Bool:5} \to \text{Bool:4} \;\vdash\; \text{x} : \text{Bool:3} \to \text{Bool:2} \to \text{Bool:1}}\ (\text{Var})$$

La procedura di collegamento tra tipi è svolta all'interno del traduttore dalla funzione link_types_internal

```
def link_types_internal(first_type:Type,
                        second_type:Type,
                        goi_clauses, polarity):

    if isinstance(first_type, Arrow):
        link_types_internal(first_type.right,
                            second_type.right,
                            goi_clauses,
                            polarity)

        link_types_internal(first_type.left,
                            second_type.left,
                            goi_clauses,
                            (not polarity))

    if isinstance(first_type, SimpleType):
        if(polarity):
            goi_clauses.append(
                    GOIQImplication(first_type.index,
                                    second_type.index))
            goi_clauses.append(
                    GOIAnsImplication(second_type.index,
                                      first_type.index))
        else:
            goi_clauses.append(
                    GOIQImplication(second_type.index,
                                    first_type.index))
            goi_clauses.append(
                    GOIAnsImplication(first_type.index,
                                    second_type.index))
```

La funzione viene invocata inizialmente con un valore di polarity = True, e attraversa gli alberi binari associati ai tipi aggiungendo a goi_clauses le nuove clausole necessarie per costruire le connessioni fra questi. polarity rappresenta il valore di polarità da associare al primo dei due tipi su cui la funzione è invocata: se link_types_internal viene richiamata su due tipi semplici, tale valore indicherà il verso dei due collegamenti "question" ed "answer" che saranno generati. Per quanto riguarda l'applicazione della funzione sui tipi complessi, è importante ricordare che negli alberi binari associati ai tipi, tutti i nodi rappresentano implicazioni logiche. Durante l'invocazione ricorsiva di link_types_internal sui sotto alberi associati ad un tipo Arrow, il valore di polarità viene invertito nel sotto albero sinistro, siccome questo rappresenta la premessa dell'implicazione.

4.3.5 Traduzione delle Regole di Inferenza

La fase principale di cui si compone il processo di traduzione consiste in un terzo attraversamento dell'albero di derivazione accettato in input avente lo scopo di creare i collegamenti associati ad ogni regola di inferenza, secondo lo schema visto nel Capitolo 3.

Nell'operazione viene impiegata la funzione link_types_internal per la generazione delle connessioni, facendo attenzione a costruire anche i collegamenti tra i tipi presenti negli environment dei termini costituenti le premesse e le conclusioni delle regole.

4.4 Un Esempio di Traduzione

Concludiamo con un esempio di traduzione, riguardante il programma:

$$\dfrac{\dfrac{\dfrac{}{\vdash \text{and} : \text{Bool:6} \to \text{Bool:5} \to \text{Bool:4}}\,(\text{and}) \quad \dfrac{}{\vdash \text{PAtom} : \text{Bool:7}}\,(\text{prob})}{\vdash \text{and PAtom} : \text{Bool:3} \to \text{Bool:2}}\,(\text{app}) \quad \dfrac{}{\vdash \text{tt} : \text{Bool:8}}\,(\text{tt})}{\vdash \text{and PAtom tt} : \text{Bool:1}}\,(\text{app})$$

Per prima cosa occorre scrivere il programma funzionale in Python, attraverso le classi che abbiamo definito:

```
A = ProbAtomRule(Expression(Environment({}),
                            TermProbAtom(),
                            Bool()))

B = AndRule(Expression(Environment({}),
                       Term("and"),
                       Arrow(Bool(), Arrow(Bool(), Bool()))))

C = ApplicationRule(Expression(Environment({}),
                    Term("and ProbabilityAtom"),
                    Arrow(Bool(), Bool())), B, A)

D = TTRule(Expression(Environment({}),
                      Term("tt"),
                      Bool()))

E = ApplicationRule(Expression(Environment({}),
                    Term("(and ProbabilityAtom) tt"),
                    Bool()), C, D)
```

L'alberò viene costruito per composizione: prima vengono definite le foglie, rappresentate dalle regole PROB ed AND, che vengono memorizzate nelle variabili Python A e B. Successivamente viene definita la regola di applicazione memorizzata in C, che ha come premesse le due foglie A e B. Si prosegue in questo modo fino ad arrivare alla variabile E, che conterrà l'intero albero da tradurre. A questo punto possiamo invocare sul nostro input E la funzione compile, che eseguirà tutte le varie fasi trattate in questo capitolo.

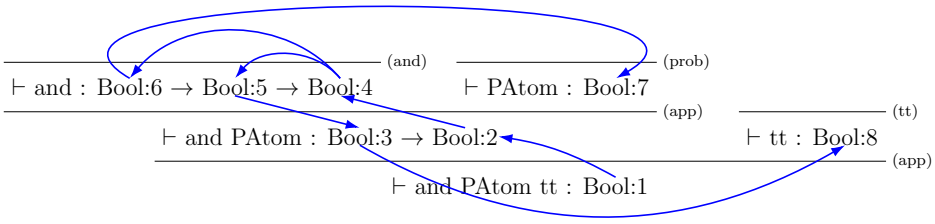
La traduzione terminerà con la generazione di un file output.pl contenente il programma ProbLog risultante:

```
%Definizione degli operatori
not(tt, ff).
not(ff, tt).
and(tt, tt, tt).
and(tt, ff, ff).
and(ff, tt, ff).
and(ff, ff, ff).
or(tt, tt, tt).
or(tt, ff, tt).
or(ff, tt, tt).
or(ff, ff, ff).

%Query ProbLog
query(ans(B,1)).

%Domanda iniziale
q(1).

%Definizione della costante probabilistica utilizzata
0.5::probatom(7).

%Codice generato dall'albero
q(2):-q(1).
ans(B,1):-ans(B,2).
q(8):-q(3).
ans(B,3):-ans(B,8).
q(4):-q(2).
ans(B,2):-ans(B,4).
q(3):-q(5).
ans(B,5):-ans(B,3).
```

```
31 q(7):-q(6).
32 ans(B,6):-ans(B,7).
33 q(5):-q(4).
34 q(6):-q(4).
35 ans(X,4):-ans(A,5), ans(B,6), and(A,B,X).
36 ans(tt,7):-q(7), probatom(7).
37 ans(ff,7):-q(7), \+probatom(7).
38 ans(tt,8):-q(8).
```

Eseguendo sull'interprete ProbLog il codice precedente, otterremo che la variabile B, rappresentante l'output del nostro programma, ha una probabilità pari a 0.5 di valere `tt` piuttosto che `ff`.

Le richieste generate fra i tipi durante l'operazione di traduzione si rispecchieranno sull'albero di derivazione nel modo seguente:



Conclusioni

In questa tesi è stato affrontato lo sviluppo di un traduttore avente un interesse sia di tipo teorico che pratico. I due linguaggi su cui opera permettono approcci totalmente differenti alla programmazione, adatti a situazioni diverse ed ognuno con i suoi specifici punti di forza. Il codice logico prodotto, in particolare, risulta interessante per via della sua struttura composta da regole estremamente semplici prese singolarmente, che sono in grado di descrivere programmi di qualsiasi complessità attraverso elementari interazioni. La rilevanza teorica del traduttore è dovuta alla sua diretta implementazione della Geometria dell'Interazione, che ci fa capire quanto potenziale risieda in un semplice cambio del punto di vista con cui osserviamo un programma. A tal proposito, esistono anche altre semantiche collegate alla Geometria dell'Interazione, di grande interesse teorico per la loro capacità di analizzare strutture come gli alberi di deduzione naturale secondo punti di vista ancora più originali ed affascinanti. Cito ad esempio la Game Semantics per la logica lineare [5], da cui sono stati presi spunti anche per la scrittura di questo elaborato.

Riguardo eventuali sviluppi futuri del traduttore, l'estensione dei costrutti disponibili nel linguaggio preso in ingresso costituisce senz'altro la direzione più interessante e dalle maggiori potenzialità.

Bibliografia

[1] Benjamin C. Pierce, Types and Programming Languages, The MIT Press, gennaio 2002.

[2] Leon S. Sterling and Ehud Y. Shapiro, The Art of Prolog, Second Edition: Advanced Programming Techniques, The MIT Press, 1994.

[3] ProbLog: Probabilistic Programming, https://dtai.cs.kuleuven.be/problog/

[4] L. De Raedt and A. Kimmig. Probabilistic (logic) programming concepts. Machine Learning, 100:1, pp. 5 - 47, Springer New York LLC, 2015.

[5] Samson Abramsky, Information, Processes and Games, Oxford University Computing Laboratory.

[6] Jean-Yves Girard. Towards a geometry of interaction. Contemporary Mathematics Volume 92, pages 69 – 108. American Mathematical Society, 1989.